Efficiëntie van raymarching in renderen

Taeke Roukema

December 2023

Samenvatting

Renderen met raymarching

Inhoudsopgave

1 Voorwoord

2 Inleiding

2.1 Introductie onderwerp

Renderen is overal. Als je je telefoon opent zie je allerlei gerenderde vormen. Bij het ontbijt zijn verpakkingen volgeprint met teksten die met de computer getekend zijn. Als je langs een bouwterrein loopt zie je hyperrealistische visualisaties van de architectuur. Moderne blockbuster-films zitten tegenwoordig bomvol CGI[1]. En er zijn al tientallen jaren films te zien die helemaal door de computer gemaakt zijn.

Voor Toy Story 3 (Figuur **??**) werd er gemiddeld zeven uur over gedaan om een frame te renderen [**?**]. En dat terwijl er gebruik werd gemaakt van twee gigantische render farms[2]. Het renderen van films kost niet alleen enorm veel tijd, maar ook veel energie. Het is dus belangrijk dat het zo efficiënt mogelijk gebeurd. Er wordt over de hele wereld voortdurend onderzoek gedaan naar manieren om dit proces efficiënter te maken en te verbeteren. De opkomst van kunstmatige intelligentie begint al bewegingen te maken in de wereld van CGI [**?**]. Maar er wordt ook voortdurend voortuitgang gemaakt op fundamentelere manieren. Zo zijn er de afgelopen vijf jaar GPU's[3] van Nvidia op de markt gekomen met ingebouwde support voor realtime raytracing[**?**]. Door op het hardware niveau de chips zo te ontwerpen dat ze heel goed zijn in bepaalde berekeningen die gebruikt worden voor het simuleren van licht kunnen GPU's gebruikt worden om voormalig minutendurende processen meer dan zestig keer per seconde uit te voeren.

Figuur 2.1: Een frame uit Toy Story 3, aan de linkerkant worden geen lichtberekeningen gedaan, en aan de rechterkant wel.

Er zijn twee belangrijke maatstaffen waarmee we de efficiëntie van een renderalgoritme kunnen meten. De eerste is vanzelfsprekend: snelheid. Als een frame sneller gerenderd is wordt er minder energie gebruikt en zijn we goedkoper uit.

[1] **C**omputer **G**enerated **I**magery

[2] Een computercluster speciaal gemaakt voor het renderen van CGI, de term was geïntroduceerd in de productie voor Bored Room[**?**]

[3] **G**raphical **P**rogramming **U**nit

Figuur 2.2: De videogame Minecraft kan gebruik maken van Nvidia GPU's om realtime lichtsimulaties te berekenen.

Maar ook geheugenbezetting is belangrijk om rekening mee te houden. Het geheugen is simpelweg de plaats in de computer waar alle informatie wordt opgeslagen. Als je berekeningen doet moet je ergens de resultaten tussendoor opslaan. Complexe scènes kunnen enorm veel details hebben, die allemaal in het geheugen opgeslagen zijn. Het is niet gratis om extra geheugen toe te voegen, het is dus belangrijk om de geheugenbezetting te minimaliseren.

2.2 Relevantie

Vrijwel alles is tegenwoordig op een manier gerenderd. Objecten zijn gedesigned met gebruik van CAD[4]. Besturingssytemen runnen op een grafische shell. En een meerendeel van advertenties gebruikt CGI.

Volgens Peter Collinridge[**?**] gebruikt de render farm van pixar 24000 processor cores verdeeld over 2000 computers. Er wordt dus waarschijnlijk gebruik gemaakt van computers met $24000/2000 = 12$ cores. De Ryzen 9 5900x is een voorbeeld van een processor met 12 cores, het energiegebruik zal niet exact hetzelfde zijn maar het ligt bij elkaar in de buurt. De 5900x gebruikt 105 Watt. $105W \cdot 2000 \approx 2,10 \cdot 10^5 W$. Volgens dezelfde bron kostte het renderen van Monster's University twee jaar. Dat is $2 \cdot 365 \cdot 24 \approx 17520h$.

$$210kW \cdot 17520h \approx 3679200kWh$$

Een gemiddeld huishouden in Nederland gebruikt 2479 kWh per jaar. Dit betekent dat het renderen van Monster's University $3679200/2479 \approx 1484$ huishoudens een jaar lang van energie had kunnen voorzien. En dat is alleen nog maar de processorkracht, er gaat ook nog energie naar de moederborden, het geheugen, de koeling en de harde schijven. Kortom, er valt een hoop te besparen.

[4] **C**omputer **A**ided **D**esign

2.3 Onderzoeksvraag/deelvragen

2.3.1 Hoofdvraag WIP

In welke situaties is raymarching een efficiëntere rendertechniek dan polygonaal renderen?

Vergelijking van twee rendermethoden in rendersnelheid en geheugenbezetting.

Vergelijking van raymarching en polygonaal renderen in rendersnelheid en geheugenbezetting.

Hoe wegen raymarching en polygonaal renderen tegen elkaar op in rendersnelheid en geheugenbezetting?

2.3.2 Deelvragen WIP

- Hoe beschrijf je een driedimensionale vorm?
- Hoe werkt raytracing?
- Hoe werkt raymarching?
- Hoe werkt rasterization?
- Hoe beschrijf je een vorm zodat het gerenderd kan worden met raymarching?
- Hoeveel geheugen neemt polygonaal renderen in?
- Hoeveel geheugen neemt renderen met raymarching in?
- Hoe snel is renderen met raymarching vergeleken met polygonaal renderen?
- Hoe kan je met raymarching objecten modelleren?

3 Theorie

3.1 Wat is renderen?

Renderen is, in feite, het weergeven van een representatie van een concept op een beeldscherm. Wij zijn voortdurend bezig met het interacteren met computers, en die interactie verloopt via het beeldscherm. Maar de computer kan uit zichzelf niet zomaar alles tekenen. Daar worden allemaal algoritmes voor geschreven. Een voorbeeld van zo'n algoritme is het tekenen van een rechthoek. In pseudocode zou je dat als volgt voor kunnen stellen:

Algorithm 1 Rechthoek Algoritme

```
1: procedure DRAWRECTANGLE(x1, x2, y1, y2)
2:     for x ← x1 to x2 do
3:         for y ← y1 to y2 do
4:             draw pixel at (x, y)
```

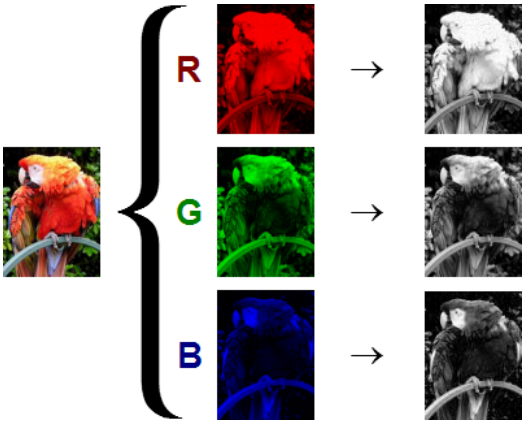
Het algoritme beschouwt elke pixel die binnen de rechthoek valt en kleurt die pixel. In dit geval wordt dat gedaan door twee loops, die samen alle mogelijke combinaties van x- en y-coördinaten doorlopen.

Renderen omvat, in principe, niks anders dan het aansturen van individuele pixels. Zo'n pixel heeft op de meeste moderne beeldschermen drie waarden die de kleur aansturen: R, G en B, die respectievelijk staan voor rood, groen en blauw. Ze kunnen een geheel getal tussen de 0 en 255 aannemen wat resulteert in 2^{24} mogelijke kleuren.

Renderen doen we op een twee-dimensionaal beeldscherm. Dat betekent dat de positie van elke pixel te beschrijven is met twee waarden. Maar de wereld om ons heen kent niet twee, maar drie ruimtelijke dimensies. Door licht dat op ons netvlies valt na weerkaatst te zijn door verschillende objecten kunnen wij die wereld representeren in onze hersenen op een tweedimensionale manier. Camera's gebruiken een gelijksoortige techniek, de lens neemt het licht en projecteert het op een sensor die de intensiteit en de kleur waarneemt. Met het renderen van driedimensionale objecten proberen we deze processen na te bootsen.

3.2 De rendervergelijking

$$L_{\mathrm{o}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) = L_{\mathrm{e}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) \; + \int_{\Omega} f_{\mathrm{r}}(\mathbf{x}, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}, \lambda, t) L_{\mathrm{i}}(\mathbf{x}, \omega_{\mathrm{i}}, \lambda, t)(\omega_{\mathrm{i}} \cdot \mathbf{n}) \, \mathrm{d}\, \omega_{\mathrm{i}}$$



Figuur 3.1: De kleuren in een foto kunnen opgesplitst worden in rode, groene en blauwe kanalen.

3.3 Wat is rasterization?

In de vroege jaren van de ontwikkeling van de computer was de rekenkracht miniscuul vergeleken met waar we vandaag de dag toegang tot hebben. Gordon Moore, mede-oprichter van Intel en legendarische informaticus, stelde in 1965 Moore's Law voor (al had het toen nog een andere naam). Moore's Law stelt dat elke twee jaar het aantal transistors in een *integrated circuit* verdubbelt [**?**]. Dit betekent dus dat 40 jaar geleden de computers $\frac{1}{2^{20}}$ keer zoveel rekenkracht hadden. Wat ongeveer één miljoenste is. Het was toen simpelweg niet mogelijk om algoritmes zoals raytracing toe te passen, omdat de complexiteit van dat soort algoritmes de capaciteit van de computers ver te boven gingen.

De oplossing was relatief simpel, in plaats van voor elke pixel te checken of er een object in de weg zit kijk je alleen maar naar de positie van de hoeken van een vorm. Als je daar vervolgens de schermcoördinaten van hebt kan je de vorm gewoon invullen, wat een vrij goedkoop proces is.

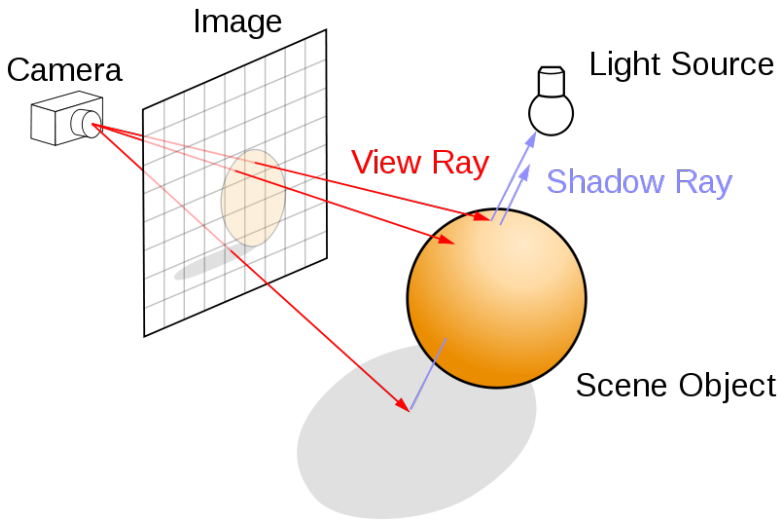
Rasterization is de standaard in de wereld van realtime renderen, vanwege het significante voordeel op de snelheid. Echter kent de methode veel nadelen. De belangrijkste van deze nadelen is het feit dat het beeld altijd een benadering zal zijn van de echte wereld. Het licht wordt niet gesimuleerd, er worden verschillende trucs gebruikt om te doen alsof het echt is maar het zal nooit echt kunnen zijn.

3.4 Wat is raytracing?

Om licht realistischer te simuleren ligt het voor de hand om vanuit elke lichtbron miljoenen lichtstralen af te vuren, die laat je door de scène kaatsen totdat ze het zichtveld van de observeerder raken. In theorie is dit de realistische benadering,

en ook exact wat de rendervergelijking op het eerste oog impliceert. Deze methode wordt *photon tracing* genoemd, naar de naam van lichtdeeltjes: fotonen. Maar in de werkelijkheid is dit vrijwel onmogelijk. Maar een klein gedeelte van de lichtstralen bereikt daadwerkelijk ooit de camera, wat betekent dat verreweg de meeste berekeningen die gemaakt worden geen invloed hebben op het uiteindelijke eindbeeld.

Om dit probleem op te lossen vuren we de stralen niet af vanuit de lichtbronnen, maar vanuit de camera. Voor elke pixel op het scherm wordt een straal door het zichtveld afgevuurd. Als die straal een object raakt kaatsen we die vervolgens af naar de lichtbronnen om te kijken hoe sterk die pixel belicht moet zijn, en of er eventueel een object in de weg zit die een schaduw zou werpen. Bovendien kunnen we de stralen door laten kaatsen tussen objecten, hiermee kunnen we een model van reflectie benaderen.

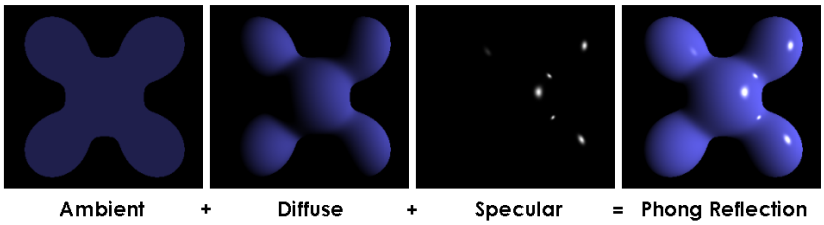


Figuur 3.2: Een diagram die laat zien hoe raytracen werkt

Het renderen wordt met deze methode al een stuk meer te overzien. Voor het genereren van een plaatje van 1000×1000 pixels vuren we een miljoen stralen af, wat klinkt alsof het veel is. Maar voor moderne computers is dat zeker haalbaar. Pas als we reflecties toe gaan passen lopen we tegen een klein probleem aan, omdat de rendertijd dan polynomiaal groeit met het aantal objecten in de scène. Dit is omdat we elk object moeten testen met ieder ander, wat zorgt voor een kwadratisch verband.

3.5 Hoe werkt shading?

Eén manier om lichtsimulaties te benaderen is het *Phong Reflection Model* [**?**]. Hierbij wordt de belichting van een object opgesplitst in drie delen: Ambient, Diffuse en Specular.



Figuur 3.3: De verschillende componenten van het *Phong Reflection Model*.

Ambient reflectie staat voor de kleine hoeveelheden licht die eigenlijk altijd wel aanwezig zijn door licht wat door de ruimte heen kaatst. Aangezien het berekenen van deze weerkaatsingen enorm veel rekenkracht kost wordt er uitgegaan van een universele gelijke belichting.

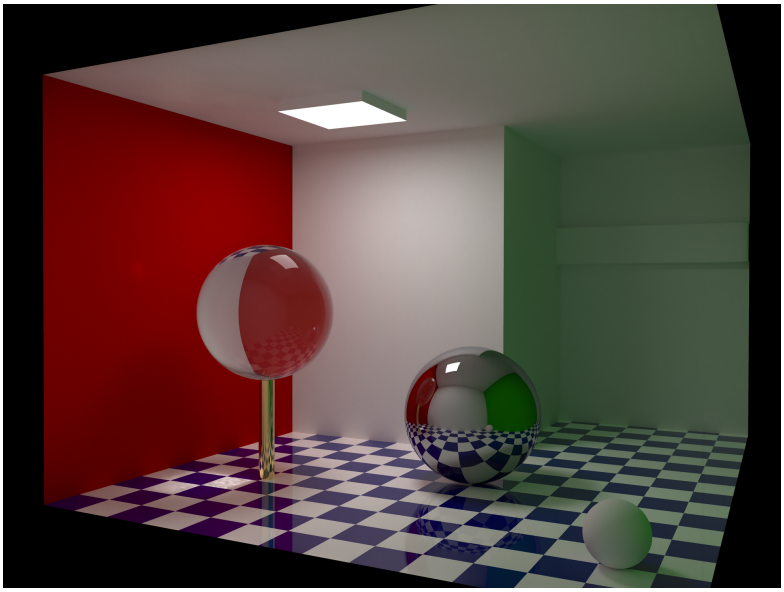
Diffuse reflectie wordt berekend aan de hand van de hoek tussen het licht op het object en de normaal. Als deze hoek groter is wordt de intensiteit kleiner, omdat het licht meer verdeeld wordt over het oppervlakte. Bij diffuse reflectie ga je ervan uit dat het licht evenveel in elke richting wordt afgekaatst, daarom wordt dit vooral veel voor ruwe oppervlakten gebruikt, waar kleine imperfecties in de textuur ervoor zorgen dat het licht alle kanten op gekaatst wordt.

Bij gladdere oppervlakten is er sprake van specular reflectie. Hier wordt het licht meer gereflecteerd in de tegenovergestelde richting dat het binnenkomt tegenover de normaal.

Als deze drie technieken samengevoegd worden krijg je een vrij accurate benadering van de beweging van echt licht.

3.6 Wat is global illumination?

De methode die hierboven beschreven staat is niet perfect. Als een straal een object ziet tussen het raakpunt en de lichtbron wordt dat punt helemaal niet belicht. Dit is omdat het algoritme niet berekent hoe het licht door de kamer heen kaatst vanuit de lichtbron. Dat concept heet *global illumination*. Het benaderen van *global illumination* is in principe hetzelfde probleem als het benaderen van de rendervergelijking. Dit betekent dat we technieken uit de wiskunde voor het numeriek oplossen van integralen direct toe kunnen passen op de rendervergelijking, waarmee we realistische beelden kunnen tekenen.



Figuur 3.4: Een plaatje gerenderd met *global illumination*.

Eén van die technieken valt binnen de categorie van Monte-Carlosimulaties. Monte-Carlosimulaties gebruiken een willekeurig element om in principe deterministische problemen op te lossen. Bij de rendervergelijking manifesteert dit zich in *path tracing*. Bij *path tracing* worden bij elk geraakt punt willekeurige nieuwe stralen afgevuurd, deze kaatsen door totdat ze een licht bereikt hebben. We komen hier bij hetzelfde probleem als eerder besproken terecht, de kans op het raken van een licht is niet erg groot. Hiertoe zijn twee belangrijke oplossingen. De eerste is *bidirectional path tracing*, hier wordt tegelijk een straal vanuit de camera afgevuurd als van het licht. Na een bepaalde hoeveelheid kaatsen worden de twee verbonden waardoor er altijd een pad van de camera naar het licht is. De andere oplossing wordt *importance sampling* genoemd. Daarbij is de distributie van de willekeurig afgekaatste lichtstralen niet uniform, er worden meer stralen richting de lichtbronnen afgekaatst. Hierdoor bereiken de stralen natuurlijk sneller de lichtbronnen. Met genoeg *samples*[5] geeft dit algoritme een bijna perfecte benadering van de rendervergelijking, dat maakt het ook de industriële standaard van het moment. Maar het is relatief inefficiënt, dit komt doordat er met te weinig *samples* veel ruis in het plaatje ontstaat. Dit is simpel te verklaren, omdat het algoritme fundamenteel willekeurig is zullen sommige pixels meer licht vinden dan die er direct naast. Puur omdat ze 'geluk' hadden. Slechts de limiet van het aantal *samples* naar oneindig zou een werkelijk antwoord geven op de rendervergelijking. Daarom is er altijd een compromis met hoeveel tijd je wil besteden aan het renderen van een

[5]Hoeveelheid stralen

frame en de kwaliteit van het resultaat. Steeds vaker wordt kunstmatige intelligente gebruikt om die ruis weg te nemen of te verminderen [**?**] maar in de industrie is dat nog steeds niet de norm.

Een andere manier om integralen op te lossen is de eindige-elementenmethode

3.7 Wat zijn polygonen?

Een polygoon is, in zijn elementairste vorm, een eindig aantal 1ijnstukken die met elkaar verbonden zijn in een polygonale lijn. Dit houdt in dat elk uiteinde van een lijnstuk het begin van een nieuwe betekent. Bovendien betekent dit dat de lijnstukken altijd één keten vormen.

Bij computer graphics hebben we het altijd over simpele polygonen. Deze polygonen snijden zichzelf niet en hebben geen gaten. Die simpele polygonen hebben een handige eigenschap, ze zijn altijd op te delen in driehoeken. Die driehoeken liggen altijd op een vlak, waar makkelijk een lijn mee te snijden valt. Vervolgens zijn er andere strategieën waarme te bepalen valt of dat punt in de driehoek ligt.

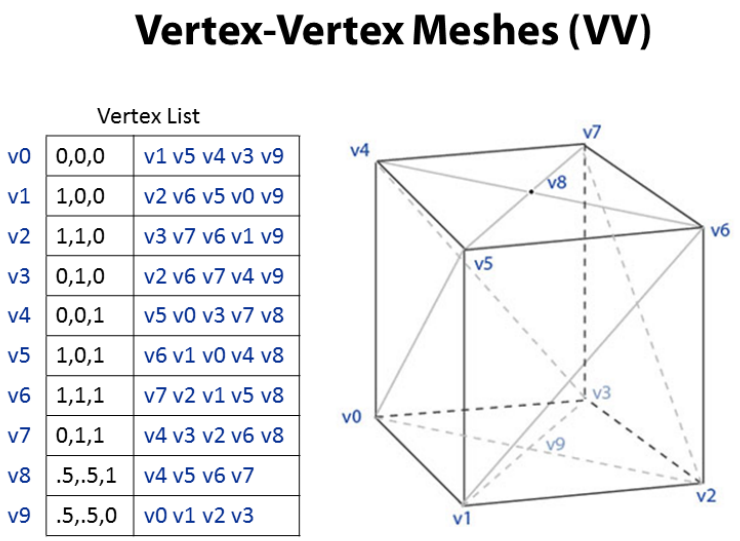
Driedimensionale vormen worden met moderne rendertechnieken vrijwel altijd beschreven met deze polygonen. De vorm wordt dan een *mesh* genoemd.

3.8 Hoe wordt een mesh opgeslagen?

Er zijn twee hoofdtechnieken: *vertex-vertex meshes* [**?**] en *face-vertex meshes*. De *vertex-vertex meshes* of VV-meshes bevatten één lijst die alle punten beschrijft. Daarnaast staan alle verbindingen die dat punt heeft met andere punten. In theorie minimaliseer je met deze techniek de geheugenbezetting. Echter maakt het het ontleden van driehoeken uit de vorm erg complex. Daarvoor zijn FV-meshes geschikter. Daar wordt van twee lijsten gebruik gemaakt. De eerste bevat weer alle punten, echter worden in plaats van de andere punten waarmee die verbonden zijn de verbonden vlakken beschreven. De andere lijst beschrijft al die vlakken en de hoekpunten ervan. De vlakken kunnen alleen driehoeken beschrijven, maar het eerder genoemde polygonale triangulatie principe betekent dat enige polygonale vorm hiermee beschreven kan worden.

3.9 Wat is raymarching?

Raymarching is een methode om het snijpunt van een straal en een vorm te vinden door iteratief stappen te zetten richting die vorm. De meest gebruikelijke methode hiervoor heet sphere tracing. Hierbij wordt gebruik gemaakt van de afstand van het originele punt tot het dichstbijzijnde van de vorm. In de richting van de straal wordt verplaatst met de afstand r tot dat dichtstbijzijnde punt. Dit wordt de locatie



Figuur 3.5: Voorbeeld van VV-representatie van een mesh.

van het nieuwe punt. Op dat punt wordt het proces weer herhaald totdat de afstand onder een van te voren bepaalde grenswaarde valt.

Algorithm 2 Raymarch Algoritme

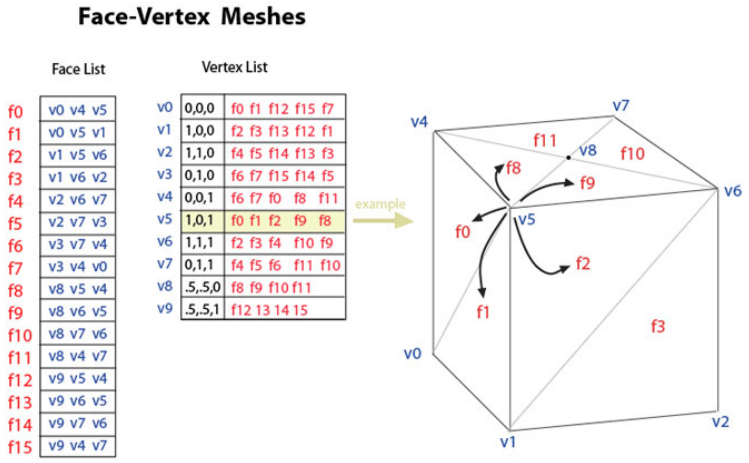
```
1: procedure MARCHRAY(pos, rot, obj, ϵ)
2:     r ← distance between pos and obj
3:     while r < ϵ do
4:         pos ← pos + rot · r
5:         r ← distance between pos and obj
       return pos
```

3.10 Wat zijn SDFs?

Het bepalen van de afstand van een punt tot het dichstbijzijnde punt op een vorm wordt gedaan met zogeheten *signed distance*-functies. *Signed* staat in deze term voor het feit dat aan de binnenkant van de vorm de functie een negatief getal geeft. De functie van een bol met middelpunt C en straal r is bijvoorbeeld als volgt:

$$\phi(x) = |x - C| - r$$

SDFs zijn enorm nuttig omdat we bewerkingen op deze formule toe kunnen passen waardoor we met vrijwel geen rekenkracht complexe vormen kunnen creëren.



Figuur 3.6: Voorbeeld van FV-representatie van een mesh.

Neem vorm ϕ_1 en vorm ϕ_2. Als je de *union*[6] wil nemen van de vormen gebruik je de volgende simpelweg de volgende formule:

$$\phi_{u(1,2)} = min(\phi_1, \phi_2)$$

De *subtraction*[7] wordt als volgt berekend:

$$\phi_{s(1,2)} = max(-\phi_1, \phi_2)$$

En de *intersection*[8] zo:

$$\phi_{i(1,2)} = max(\phi_1, \phi_2)$$

Dit worden de boolean operaties genoemd, en staan respectievelijk voor: OR, NOT en AND.

Een andere interessante operatie is de modulus, hiermee is het mogelijk om een vorm voor eeuwig te laten herhalen. Voor een periode c en een SDF ϕ is de formule hiervoor:

$$\phi_{mod} = \phi + \frac{1}{2}c \mod c$$

[6]De vormen samengevoegd
[7]De ene vorm van de andere afgetrokken
[8]Het deel wat bij beide vormen hoort

4 Hypothese

5 Ontwikkeling

5.1 Hardware

5.2 Software

5.2.1 Besturingssysteem

Het uitvoeren en programmeren van de code zal volledig met Linux gedaan worden. Deze keuze is om meerdere redenen gemaakt. Ten eerste is Linux de standaardkeuze voor developers om te ontwikkelen. Windows en MacOS zijn ontwikkeld als product voor de gebruiker, terwijl Linux ontwikkeld is voor stabiliteit en betrouwbaarheid. Dit heeft als effect dat er met Linux veel minder tegen het besturingssysteem in gewerkt hoeft te worden. Bovendien zijn alle tools die gebruikt worden voor dit onderzoek FOSS[9], en daardoor voor zowel Windows als Linux beschikbaar. Dus qua support maakt het geen verschil.

Als distributie[10] is voor Manjaro[11] gekozen. Dit is een fork van Arch Linux. Arch Linux staat erom bekend dat het altijd de nieuwste versie van software support. Dit komt doordat het een *rolling release model* heeft, in tegenstelling tot *fixed release*, waar distributies zoals Debian gebruik van maken. Bovendien heeft Arch Linux toegang tot de Arch User Repository (AUR). Dat is een enorme collectie van software die gebruikers zelf kunnen uploaden naar de Arch servers, met helpers zoals *yay* kan je met één commando vrijwel alle software die beschikbaar is op GNU/Linux op de computer installeren. De combinatie van deze voordelen maken Arch Linux een erg aantrekkelijke distributie om te gebruiken als programmeur. Manjaro voegt de rest van de functies toe aan het besturingssysteem, zoals een Desktop Environment (DE) en een terminal.

5.2.2 Programmeertaal

Er zijn talloze programmmeertalen die geschikt zijn voor graphics programming. Daarom was de keuze voor de programmeertaal niet makkelijk. Zelf heb ik al jarenlang ervaring met Python[12], maar deze taal staat niet bekend om de snelheid. Dit komt doordat het een *interpreted* taal is. Dat betekent dat de code live gelezen wordt wanneer gerund. Dit staat tegenover *compiled* talen, die de code eerst compileren naar machinetaal. Die machinetaal is veel efficiënter te lezen door computers, waardoor de snelheid toeneemt. Een andere optie was Javascript, het grote

[9] **F**ree and **O**pen **S**ource **S**oftware

[10] Linux zelf is slechts een *kernel* die de interactie tussen de hardware en de software regelt. Bovenop deze *kernel* bestaan distributies die het een werkend besturingssysteem maken.

[11] https://manjaro.org/

[12] https://www.python.org/

voordeel van deze taal is dat hij speciaal voor het web gemaakt is. Hierdoor zou het delen van het gemaakte project met anderen zo simpel zijn als het doorsturen van een url. Bovendien maakt Javascript op moderne browsers gebruik van Just In Time (JIT) compilation. Dat is een combinatie tussen *interpreted* en *compiled* waar de code live omgezet wordt in machinetaal voordat het gerund wordt. Maar toch is zelfs Javascript niet snel genoeg. Bovendien missen beide talen iets wat erg belangrijk is in computer graphics: controle over het geheugen. Scènes kunnen enorm complex zijn dus het is belangrijk dat die zo efficiënt mogelijk in het geheugen geplaatst worden, en het geheugen moet weer gewist worden wanneer het niet meer gebruikt wordt. Python en Javascript geven allebei niet die controle, in plaats daarvan probeert de *interpreter* zelf zo efficiënt mogelijk het geheugen te gebruiken. Om deze redenen heb ik gekozen voor C++, deze taal is in 1985 uitgevonden door Brane Stroustrup en wordt vandaag de dag nog door 20,17% van Stack Overflow gebruikers gebruikt[**?**]. De taal is ontwikkeld als extensie voor C, waardoor het moderne functies heeft zoals *Object Oriented Programming* (OOP) en datastructuren. Maar het heeft tegelijkertijd alle voordelen die C heeft als low-level taal.

5.2.3 Framework

C++ heeft uit zichzelf nog geen grafische capabiliteiten. Daar is een framework voor nodig. Moderne grafische kaarten zijn allemaal gemaakt met speciale specificaties, die ervoor zorgen dat het besturingssysteem weet hoe hij moet communiceren met de GPU. Er zijn verschillende van deze specificaties met verschillende doelen. Zo heb je DirectX, die specifiek gemaakt is voor Windows. En wat algemenere API[13] is OpenGL (Open Graphics Library). Met C++ is het dan ook mogelijk om direct gebruik te maken van deze API, net als in de meeste programmeertalen.

Maar toch heb ik daar niet voor gekozen. Dit is omdat OpenGL heel goed is in het implementeren van bestaande rendermethodes, waar de GPU ook voor ontwikkeld is. Dit maakt het ideaal voor het bouwen van videogames, omdat het daar heel snel in is. Maar minder voor dit specifieke onderzoek. Ik wil objectief vergelijken hoe de verschillende rendermethoden tegen elkaar opwegen, als de gebruikelijkere methodes heel goed geoptimaliseerd zijn door de GPU en OpenGL zou dat oneerlijk zijn en de data onbetrouwbaar maken. Daarom heb ik gekozen voor raylib[14]. Raylib is een zeer minimalistische *library* die alle basistools geven die we nodig hebben om te kunnen tekenen op een canvas, terwijl het tegelijkertijd razendsnel blijft.

[13] **A**pplication **P**rogramming **I**nterface
[14] https://www.raylib.com/

5.2.4 Integrated Development Environment

Ik ga al mijn programmeren doen in Visual Studio Code[15] omdat het een mooie simpele text editor is die precies doet wat ik wil. Het geeft goede IntelliSense[16], syntax highlighting en het geeft een goed overzicht van het project. Bovendien heeft het een enorme markt van plugins die het product nog meer verbeteren. Zo gebruik ik de Vim keybinds plugin om de efficiënte workflow van de editor Vim[17] te emuleren.

5.3 Programmeren

5.3.1 Specificaties

Voor het onderzoek ga ik twee verschillende renderers ontwikkelen. De eerste gebruikt raytracing, en de objecten worden beschreven met raymarching. De tweede gebruikt ook raytracing alleen worden de objecten polygonaal beschreven. Het is voor het onderzoek belangrijk dat de twee algoritmes zo veel mogelijk op elkaar lijken, zodat het vergelijkende onderzoek iets objectiefs kan zeggen over de effectiviteit van de verschillende technieken. Vandaar dat hier eerst de specificaties worden beschreven waar de algoritmes aan zullen moeten doen. Deze specificaties kunnen uitgebreid worden of ingekort gebaseerd op de progressie over de tijd.

De renderer moet kunnen renderen vanuit een **camera** die een bepaalde positie heeft binnen de driedimensionale ruimte. De camera heeft ook een *field of view* (fov), die aangeeft hoe breed het zichtveld is. Uit de camera komt uit elke pixel een *ray* die snijdt met de *viewport*. De resolutie van het beeldscherm wordt in het programma aangegeven. Waarschijnlijk willen we renderen op een resolutie van 500×500. De *rays* worden beschreven als parametrische vergelijking. Hierdoor kan de parameter t gebruikt worden om de straal te snijden met de objecten. De objecten zijn uiteraard anders beschreven in de twee verschillende programma's, maar ze hebben wel delen gemeen. Zo heeft een object altijd een positie, een draaiing en een grootte. Verder wordt het materiaal van elk object beschreven met een kleur, de *specularity* en het reflectievermogen. De kleur wordt beschreven met RGBA, met waarden op $[0, 255]$. De *specularity* geeft de macht waarmee de cosinus die de verspreiding van de *highlights* beschrijft wordt verheven. Het reflectievermogen is een getal op $[0, 1]$ die het aandeel van de kleur beschrijft die komt uit reflecties. Bovendien willen we een aantal primitieve vormen hebben die standaard beschikbaar zijn. Voor raymarching zal dit triviaal zijn aangezien elke

[15] https://code.visualstudio.com/

[16] Verzamelnaam voor tools die helpen in het schrijven van code zoals: code completion en informatie over parameters

[17] https://www.vim.org/

vorm slechts een *signed distance function* is. Maar voor het polygonale renderen moeten hier nog aparte algoritmes voor komen. De vormen die ondersteund moeten worden zijn als volgt:

- Bol
- Vlak
- Driehoek
- Balk
- Ellipsoïde
- Cylinder

Verder zijn er ook verschillende soorten lichten mogelijk. Die nemen drie typen aan:

- Ambient
- Point
- Directional

Alle lichten hebben een intensiteit, die aangeeft hoe vel dat licht is. Ambient lichten belichten de hele scène met een standaard hoeveelheid, er wordt hierbij dus geen rekening gehouden met schaduwen. Ambient lichten hebben maar één waarde: de intensiteit, die een waarde op $[0, 1]$ aanneemt. Point lichten hebben daarbij ook nog een positie, bovendien kan de intensiteit hoger dan 1 zijn. Directional lichten hebben geen positie maar een richting, en hebben ook een intensiteit die hoger kan zijn dan 1.

De camera, alle objecten en de lichten worden allemaal beschreven met een Scene() object. Die heeft de methodes AddSDFObject(), AddPolygon(), AddPolygonalPrimitive() en AddLight(). De scène zal in een los bestand van de rest van de code beschreven worden.

5.3.2 Raylib uitproberen

5.3.3 Computer Graphics From Scratch

Voor het bouwen van de raytracer is gebruik gemaakt van het boek *Computer Graphics From Scratch* [**?**] van Gabriel Gambetta. Het boek beschrijft stap voor stap hoe men een renderer kan bouwen, de eerste helft gaat over een raytracer en de

tweede helft een rasterizer. Het geeft een duidelijke beschrijving van de wiskundige theorie achter raytracen en stelt met pseudocode een structuur voor waarmee het geprogrammeerd kan worden. Aan het einde heb je een programma die kan raytracen met diffuse belichting, specular highlighting, schaduwen en reflecties. Het boek beschrijft alleen maar hoe bollen getekend kunnen worden, dus de hele implementatie van de raymarching zal zelf bedacht moeten worden. Daarbovenop is er ook geen implementatie van polygonen.

5.3.4 Raytracing

De stralen worden aangegeven met een paramatervergelijking in de vorm:

$$P = O + t(V - O)$$

Hier is P enig punt in de straal van O naar V. t geeft aan hoe ver op de straal punt P is.

$(V - O)$ is gelijk aan de richting van de straal, dus die noemen we $\overrightarrow{D}$. Dan krijg je:

$$P = O + t\overrightarrow{D}$$

In het begin zal de raytracer alleen bollen kunnen tekenen, de vergelijking van punt P op een bol met middenpunt C en straal r is:

$$|P - C| = r$$

De lengte van een vector is gelijk aan de wortel van het inwendig product met zichzelf. Het inwendig product wordt aangegeven als $\langle \overrightarrow{V}, \overrightarrow{V} \rangle$. Dat geeft:

$$\langle P - C, P - C \rangle = r^2$$

Dit combineren met de vergelijking van de straal $P = O + t\overrightarrow{D}$ geeft:

$$\langle O + t\overrightarrow{D} - C, O + t\overrightarrow{D} - C \rangle = r^2$$

$O - C$ kunnen we opschrijven als $\overrightarrow{CO}$:

$$\langle CO + t\overrightarrow{D}, \overrightarrow{CO} + t\overrightarrow{D} \rangle = r^2$$

Inwendige producten zijn bilineair, dit betekent dat:

$$\langle x + y, z \rangle = \langle x, z \rangle + \langle y, z \rangle$$

En:

$$\langle x, y + z \rangle = \langle x, y \rangle + \langle x, z \rangle$$

En:

$$\langle x, \lambda y \rangle = \lambda \langle x, y \rangle = \langle \lambda x, y \rangle$$

Hieruit kunnen we de vergelijking afsplitsen tot:

$$\langle \overrightarrow{CO}, \overrightarrow{CO} \rangle + \langle t\overrightarrow{D}, \overrightarrow{CO} \rangle + \langle \overrightarrow{CO}, t\overrightarrow{D} \rangle + \langle t\overrightarrow{D}, t\overrightarrow{D} \rangle = r^2$$

Verder zijn kruisproducten commutatief dus $\langle x, y \rangle = \langle y, x \rangle$.
Hieruit volgt:

$$\langle t\overrightarrow{D}, t\overrightarrow{D} \rangle + 2\langle t\overrightarrow{D}, \overrightarrow{CO} \rangle + \langle \overrightarrow{CO}, \overrightarrow{CO} \rangle = r^2$$

Met de derde bilineairiteitsregel kunnen we dit weer omschrijven naar:

$$t^2 \langle \overrightarrow{D}, \overrightarrow{D} \rangle + t \left(2\langle \overrightarrow{D}, \overrightarrow{CO} \rangle \right) + \langle \overrightarrow{CO}, \overrightarrow{CO} \rangle - r^2 = 0$$

Nu heeft het de vorm aangenomen van de kwadratische vergelijking, die we op kunnen lossen met de ABC-formule.

$$\{t_1, t_2\} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Hierbij geldt dus:

$$a = \langle \overrightarrow{D}, \overrightarrow{D} \rangle$$

$$b = 2\langle \overrightarrow{CO}, \overrightarrow{D} \rangle$$

$$c = \langle \overrightarrow{CO}, \overrightarrow{CO} \rangle - r^2$$

In de code wordt dit als volgt geschreven:

```
Vector2 IntersectRaySphere(Vector3 O, Vector3 D, Sphere
   sphere) {
    double r = sphere.radius;
    Vector3 CO = subtract(O, sphere.center);

    double a = dot(D, D);
    double b = 2 * dot(CO, D);
    double c = dot(CO, CO) - r*r;

    double discriminant = b*b-4*a*c;

    if (discriminant < 0) {
        return (Vector2){1E9, 1E9};
    }

```

```
    double t1 = (-b + sqrt(discriminant))/(2*a);
    double t2 = (-b - sqrt(discriminant))/(2*a);

    return (Vector2){t1, t2};
}
```

Voor de situatie zonder oplossing wordt $\{10^9, 10^9\}$ teruggegeven. Later wordt gecheckt of de teruggeven waarde boven een bepaalde grens ligt, waarna hij wordt weggeworpen.

```
for (int x = -canvas.width/2; x < canvas.width/2; x++) {
    for (int y = -canvas.height/2; y < canvas.height/2; y
   ++) {
        Vector3 D = vp.CanvasToViewport(canvas, x, y);
        Color color = TraceRay(O, D, 1, 1E9, scene, 3);
        canvas.PutPixel(x, y, color);
    }
}
```

Hiervoor wordt gebruik gemaakt van de viewport, die als volgt is gedefiniëerd:

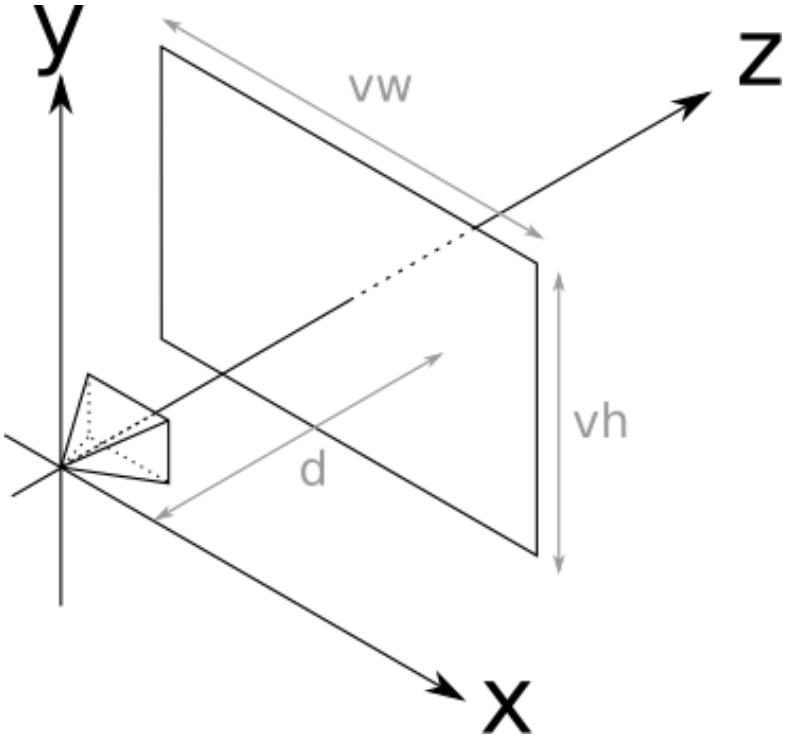
```
class Viewport {
    public:
        double width;
        double height;

        double projection_plane_distance;

        Viewport(double w, double h, double ppd);
        Viewport();

        Vector3 CanvasToViewport(Canvas canvas, int x, int
   y);
};
```

De viewport is een virtueel vlak die op een bepaalde afstand d van de camera staat. De stralen worden afgevuurd door deze viewport. Uit een punt P op de viewport kan de vector $\overrightarrow{OP}$ gehaald worden. Dit is hetzelfde als de $\overrightarrow{D}$ uit de parametervergelijking van de straal.

De volgende functie converteert screenspace coördinaten naar de coördinaten op de viewport.

```
Vector3 Viewport::CanvasToViewport(Canvas canvas, int x,
   int y) {
    return (Vector3){x*width/canvas.width, y*height/canvas.
   height, projection_plane_distance};
```

```
}
```



Figuur 5.1: Projectie van camera op viewport.

Het afvuren van de stralen vanuit de viewport wordt gedaan met de TraceRay() functie.

```
Color TraceRay(Vector3 O, Vector3 D, double t_min, double
   t_max, Scene scene) {
    std::pair<Sphere, double> closest_intersection =
   ClosestIntersection(O, D, t_min, t_max, scene);
    Sphere closest_sphere = closest_intersection.first;

    if (closest_t==1E9) {
        return RAYWHITE;
    }

    return closest_sphere.material.color;
}
```

De functie geeft een kleur terug, in dit vroegste geval is dat gewoon de kleur van het object. Die kleur wordt vervolgens voor de bijbehorende pixel getekend.

Er wordt gebruik gemaakt van de ClosestIntersection() functie.

```
std::pair<Sphere, double> ClosestIntersection(Vector3 O,
   Vector3 D, double t_min, double t_max, Scene scene) {
    double closest_t = 1E9;
    Sphere closest_sphere;

    for (Sphere sphere: scene.spheres) {
        Vector2 t_points = IntersectRaySphere(O, D, sphere)
   ;
        double t1 = t_points.x;
        double t2 = t_points.y;

        if ((t1 > t_min && t1 < t_max)&&(t1<closest_t)) {
            closest_t = t1;
            closest_sphere = sphere;
        }

        if ((t2 > t_min && t2 < t_max)&&(t2<closest_t)) {
            closest_t = t2;
            closest_sphere = sphere;
        }
    }

    return std::make_pair(closest_sphere, closest_t);
}
```

Deze geeft terug welke bol het dichst bij is, en hoe ver die bol precies is.

Samen geeft dit de eerste versie van de raytracer. We initialiseren de scène als volgt:

```
Viewport vp(1, 1, 1);

Scene scene(vp);

scene.AddSphere(Sphere(
    (Vector3){0, -1, 3},
    1,
    ObjectMaterial (
        (Color){255, 0, 0, 255},
        500,
        0.2
    )
));

scene.AddSphere(Sphere(
```

```
    (Vector3){2, 0, 4},
    1,
    ObjectMaterial (
        (Color){0, 0, 255, 255},
        500,
        0.2
    )
));

scene.AddSphere(Sphere(
    (Vector3){-2, 0, 4},
    1,
    ObjectMaterial (
        (Color){0, 255, 0, 255},
        500,
        0.2
    )
));

scene.AddSphere(Sphere(
    (Vector3){0, -5001, 0},
    5000,
    ObjectMaterial (
        (Color){255, 255, 0, 255},
        500,
        0
    )
));
```

In het begin wordt een Scene() object aangemaakt. Daaraan kunnen vervolgens objecten aan toegevoegd worden. Dat Scene() object ziet er zo uit:

```
class Scene {
    public:
        Viewport vp;

        std::vector<Sphere> spheres;

        Scene(Viewport viewport);

        void AddSphere(Sphere sphere);
};
```

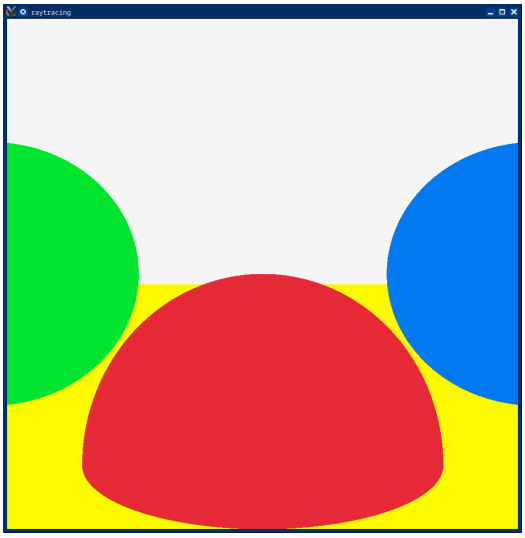
Alle bollen worden in een vector opgeslagen. Die bollen zijn zelf weer een aparte klasse, die als volgt is gedefiniëerd:

```
class Sphere {
    public:
        Vector3 center;
        double radius;

        ObjectMaterial material;

        Sphere(Vector3 center, double radius,
   ObjectMaterial material);
        Sphere();
};
```

Ieder object heeft een ObjectMaterial, in deze eerste versie beschrijft die alleen de kleur:

```
class ObjectMaterial {
    public:
        Color color;

        ObjectMaterial(Color color);
        ObjectMaterial();
};
```

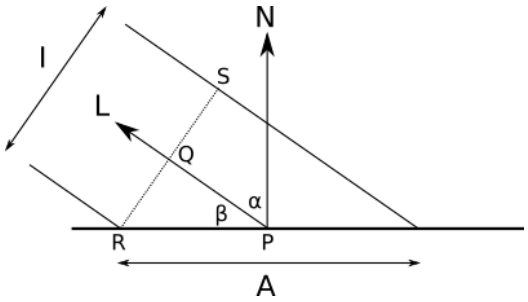
Hierna wordt de eerder beschreven loop over elke pixel gerund, hierdoor krijgen we onze eerste render.



Figuur 5.2: De eerste render gemaakt met de raytracer.

5.3.5 Diffusion

Diffusie in het Phong-Model.



Figuur 5.3: Diagram die laat zien hoe licht op een oppervlakte terecht komt.

In dit diagram raakt lichtstraal $\overrightarrow{L}$ het object in P met hoek α tot normaal $\overrightarrow{N}$. De lichtstraal heeft intensiteit I, en is verspreidt over oppervlakte A. Dit betekent dat de intensiteit per eenheid oppervlakte $\frac{I}{A}$ is. In een rechte driehoek geldt:

$$\cos\alpha = \frac{A}{S}$$

Hier is A de aanliggende zijde en S de schuine zijde. In het bovenstaande diagram is dat gelijk aan $\frac{I}{A}$.

Ook geldt dat:

$$\cos\alpha = \frac{\langle\overrightarrow{N},\overrightarrow{L}\rangle}{|\overrightarrow{N}||\overrightarrow{L}|}$$

Dus:

$$\frac{I}{A} = \frac{\langle\overrightarrow{N},\overrightarrow{L}\rangle}{|\overrightarrow{N}||\overrightarrow{L}|}$$

Hieruit kunnen we de volgende algemene formule voor diffusie opstellen, met een som voor elke lichtbron met intensiteit I_i en richting $\overrightarrow{L_i}$. Verder is I_A de ambient belichting.

$$I_P = I_A + \sum_{i=l}^{n} I_i \frac{\langle\overrightarrow{N},\overrightarrow{L_i}\rangle}{|\overrightarrow{N}||\overrightarrow{L_i}|}$$

De normaal van de bol met middelpunt C in punt P wordt zo bepaald:

$$\overrightarrow{N} = \frac{P - C}{|P - C|}$$

Door te delen met $|P-C|$ wordt de vector genormaliseerd, waardoor de lengte 1 wordt.

De volgende functie wordt gebruikt voor het berekenen van het licht:

```
double ComputeLighting(Vector3 P, Vector3 N, Vector3 V,
   Scene scene) {
    double i = 0.0;
    Vector3 L;

    for (Light light: scene.lights) {
        if (light.type == 0) {
            i += light.intensity;
        } else {
            if (light.type == 1) {
                L = subtract(light.position, P);
            } else if (light.type == 2) {
                L = light.direction;
            }

            // Diffuse
            double n_dot_l = dot(N, L);
            if (n_dot_l > 0) {
                i += light.intensity * n_dot_l/magnitude(L)
   ;
            }
        }
    }
    return i;
}
```

P is het punt op de vorm, N de normaal en L de richting van het licht. Aan het Scene() object is nu ook een vector van de lichten in de scène toegevoegd. Die lichten zijn gedefiniëerd als volgt:

```
class Light {
    public:
        int type; // 0: ambient; 1: point; 2: directional
        double intensity;

        Vector3 position;
        Vector3 direction;

        Light();
        Light(int t, double i);
        Light(int t, double i, Vector3 pos_or_rot);
```

```
};
```

De functie heeft het argument *t*, wat voor type staat. Er zijn drie types: ambient, directional en point. Die staan voor 1, 2 en 3 respectievelijk. Het initialiseren van een ambient licht gaat met de volgende functie:

```
Light::Light(int t, double i) {
    type = t;
    intensity = i;
    position = (Vector3){0, 0, 0};
    direction = (Vector3){0, 0, 0};
}
```

En voor rotationele en positionele lichten gaat het zo:

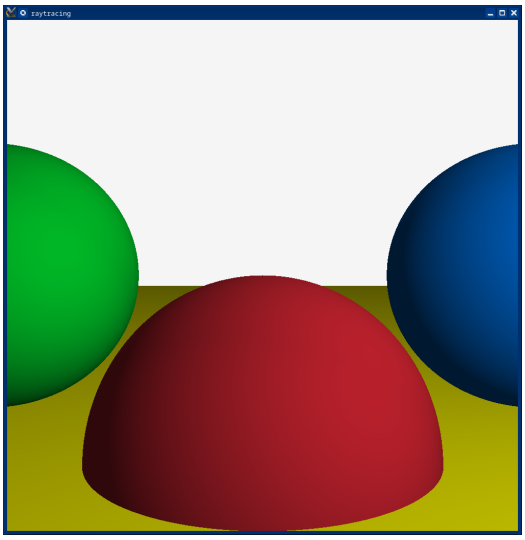
```
Light::Light(int t, double i, Vector3 pos_or_rot) {
    type = t;
    intensity = i;
    if (type == 1) {
        position = pos_or_rot;
        direction = (Vector3){0, 0, 0};
    } else if (type == 2) {
        position = (Vector3){0, 0, 0};
        direction = pos_or_rot;
    }
}
```

We voegen aan de scène de volgende lichten toe:

```
scene.AddLight(Light(
    0, // Ambient
    0.2
));

scene.AddLight(Light(
    1, // Point
    0.6,
    (Vector3){2, 1, 0}
));

scene.AddLight(Light(
    2, // Directional
    0.2,
    (Vector3){1, 4, 4}
));
```

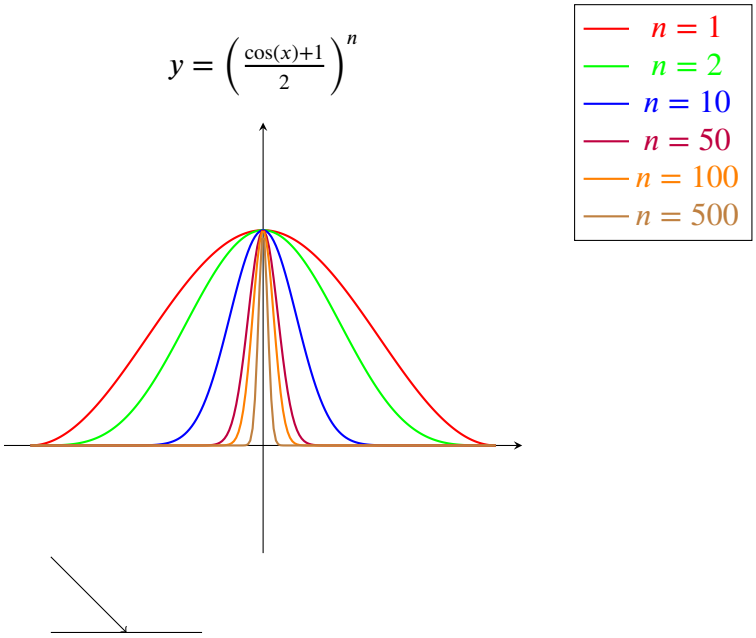
In de TraceRay() functie vermenigvuldigen we nu de originele kleur met de teruggegeven intensiteit.

```
Color TraceRay(Vector3 O, Vector3 D, double t_min, double
   t_max, Scene scene) {
    std::pair<Sphere, double> closest_intersection =
   ClosestIntersection(O, D, t_min, t_max, scene);
    Sphere closest_sphere = closest_intersection.first;
    double closest_t = closest_intersection.second;

    if (closest_t==1E9) {
        return RAYWHITE;
    }

    Vector3 P = add(O, multiply(D, closest_t));
    Vector3 N = subtract(P, closest_sphere.center);
    N = multiply(N, 1/magnitude(N));
    return multiply(closest_sphere.color, ComputeLighting(P
   , N, multiply(D, -1), scene));

}
```



Figuur 5.4: De renderer, nu met diffuse oppervlakten.

5.3.6 Specular Highlights



5.3.7 Schaduwen

5.3.8 Reflecties

5.3.9 Raymarching

5.3.10 SDFs

5.3.11 SDF operaties

5.3.12 Driehoeken

5.3.13 Mesh Representatie

5.3.14 Polygonale primitieven

5.3.15 Translatie, rotatie en schaling

5.3.16 Radiosity

5.3.17 Interface

5.3.18 Meten

6 Methode

6.1 Variabelen

6.2 Meetmethoden

7 Resultaten

7.1 Snelheid

7.2 Geheugenbezetting

7.3 Renders

8 Nauwkeurigheidsanalyse

9 Conclusie

10 Discussie

11 Nawoord

Bedankt aan mijn moeder Arria Gosman. Bedankt aan Ina, voor haar mentale steun.

12 Literatuurlijst

Referenties

[Sci, 2016] (2016). Rendering.

[Sta, 2022] (2022). Stack overflow 2022 developer survey.

[Alwani, 2018] Alwani, R. (2018). Microsoft and nvidia tech to bring photorealistic games with ray tracing.

[Anderson, 2021] Anderson, M. (2021). Nerf moves another step closer to replacing cgi.

[Chaitanya et al., 2017] Chaitanya, C. R. A., Kaplanyan, A. S., Scheid, C., Salvi, M., Lefohn, A., Nowrouzezahrai, D., and Aila, T. (2017). Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder.

[Clay, 1990] Clay, J. (1990). Making of bored room (production).

[Gambetta, 2021] Gambetta, G. (2021). *Computer Graphics from Scratch*. No Starch Press.

[Lehrer, 2010] Lehrer, J. (2010). How toy story 3 was made.

[Moore, 1965] Moore, G. (1965). Cramming more components onto integrated circuits.

[Phong, 1975] Phong, B. T. (1975). Illumination for computer generated images.

[Smith, 2006] Smith, C. (2006). On vertex-vertex systems and their use in geometric and biological modelling.

13 Logboek

Activiteit	Datum	Tijd (m)	Totale tijd (h)	% Voltooid
Programmeren	20220906	45	0.8	0.94%
Programmeren	20220908	30	1.3	1.56%
Gesprek met begeleider	20220909	20	1.6	1.98%
Programmeren	20220921	90	3.1	3.85%
Inhoudsopgave Opzet	20220928	35	3.7	4.58%
Schrijven Theorie/Achtergrond Renderen	20220930	45	4.4	5.52%
Opzetten LaTeX Document	20221002	120	6.4	8.02%